AriaDB: A key-value store in Haskell

Abhilash Kumar, Proneet Verma and Saurav Kumar
Indian Institute of Technology Kanpur
{abhilak, proneetv, ksaurav} @iitk.ac.in

Abstract

Key-value stores are an essential component in the industry, including social networks and online retail. Apart from performance, today we need concise, elegant, concurrent, type-safe and maintainable code. We present **ariaDB**, a persistent key-value store written in Haskell and a library to use the service in Haskell language. The datastore uses B+ tree to index the data for faster reads and writes.

Introduction

The aim was not to implement another key value store, but to implement a key-value store in Haskell. Infact, this key-value data store is inspired from NoSQL databases which are based on the idea of a B+ Tree indexed structure[2].

Haskell is a purely functional, statically typed and lazy language. Writing a program in Haskell requires thinking functionally and is quite different from programming in imperative languages like C++/Java.

Design

AriaDB is designed to have an uncoupled service and client utility. AriaDB server has a datastore which is indexed using B+ tree, and it exposes REST APIs to talk to AriaDB clients. In addition to the service, we have also implemented the client library for Haskell language, which can easily be extended over to any number of languages.

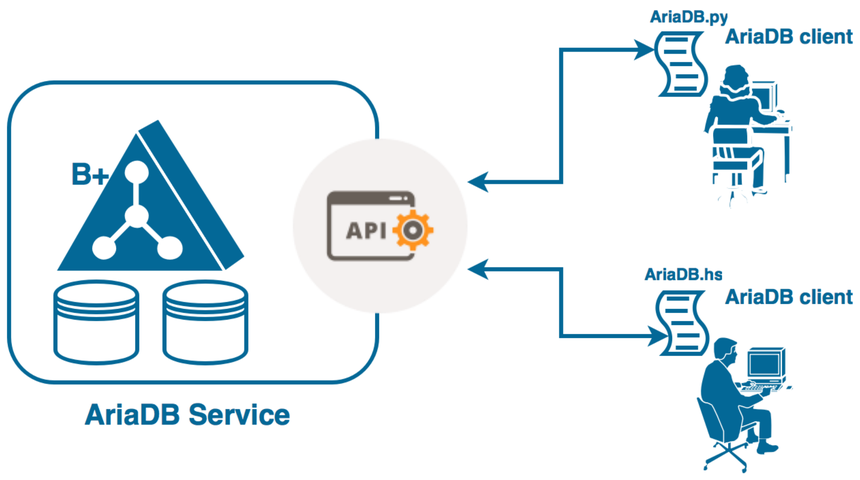

Figure 1: Design of AriaDB: separate server module and client libraries

Server

This is the HTTP API layer of the database and serves the endpoints to which the clients talk to exchange data. This is where the requests like GET, PUT and DELETE are handled. Irregular behavior is error-handled here itself.

B+ Tree

B+ Trees are primarily used here to keep data indexed based on their keys, thereby enhancing the retrieval of data. Its a disk based index structure which in turn powers up our datastore to be persistent, since in-memory structures have size restrictions.

Caching

Since our index structure is a disk based one, concurrent disk IO is one of the biggest hindrance to the performance of our database. Hence, we have implemented a in-memory caching layer to avoid repeated disk IOs and thereby improving read speed by a substantial factor.

Client

This is the interface to ariaDB server with which the user interacts with the datastore. This module basically packs the HTTP Request and sends to the server. Hence, the way our database interaction is implemented, clients may interact with the database using any programming language having an HTTP module.

The heart of the datastore is in Haskell utilizing all possible functional programming style, and powering the data with concurrent access, faster retrieval and caching but the user can interact with it using any style of programming with the only requirement that it must have an HTTP module.

Features & their implementation

Indexing:
Implemented a B+ Tree index structure to index the **AriaValue** based on the **AriaKey**.

Uncoupled Service and Library:
Using the Haskell **Warp** module, we have implemented REST APIs which responds to clients HTTP requests and thereby serving the interface layer between the client and the datastore.

Caching:
This layer is based on a **Least-Recently-Used Caching** model. Least frequently queried keys are highly likely to be removed from the in-memory store and replaced by newer keys.

Pseudo Code

```haskell
-- Types
type AriaKey    = String
type AriaValue  = String
data AriaKV = AriaKV {
  key :: AriaKey,
  value :: AriaValue
 } deriving (Show, Read)

-- Client Library
get :: (Read a) => AriaKey
        -> IO (Maybe a)
put :: (Show a) => AriaKey -> a
        -> IO ()
delete :: AriaKey -> IO ()

-- Request Handling at Warp Server
app req respond =
case requestMethod req of
 "GET" -> case pathInfo req of
 [key] -> do
  value <- Cache.get lruCache key
  case value of
   Just v  -> respond200With v
   Nothing -> respond notFound
  _ -> respond forbidden

 "POST" -> case pathInfo req of
  [key] -> do
   value <- requestBody req
   Cache.upsert lruCache key value
   respond200With ""
   _ -> respond forbidden

 "DELETE" -> case pathInfo req of
  [key] -> do
   Cache.remove lruCache key
   respond200With ""
   _ -> respond forbidden
_ -> respond forbidden
```

```haskell
-- B Plus Tree
get :: AriaKey -> IO (Maybe AriaValue)
upsert :: AriaKV -> IO ()
remove :: AriaKey -> IO ()

--Cache
get :: IORef (LRU AriaKey AriaValue)
 -> AriaKey -> IO (Maybe AriaValue)
upsert :: IORef (LRU AriaKey AriaValue)
 -> AriaKey -> AriaValue -> IO ()
remove :: IORef (LRU AriaKey AriaValue)
 -> AriaKey -> IO ()

 -- Usage
 import AriaDB
 data Person = Person {
   firstName :: String,
   lastName  :: String
 } deriving (Show, Read, Eq)

 foo = Person "John" "Doe"
 main = do
   let testKey = "k1"
   put testKey foo

   bar <- get testKey
   case bar of   -- Type Assigned
       Just v  -> print (v::Person)
       Nothing -> print "Nothing"

   baz <- get testKey
   case baz of   -- Type Inferred
     Just v -> print $ v == foo
     Nothing -> print "Not found"

   delete testKey
   qux <- get testKey
   case qux of    -- Type Assigned
     Just v  -> print (v::Person)
     Nothing -> print "Nothing"
```

Analysis

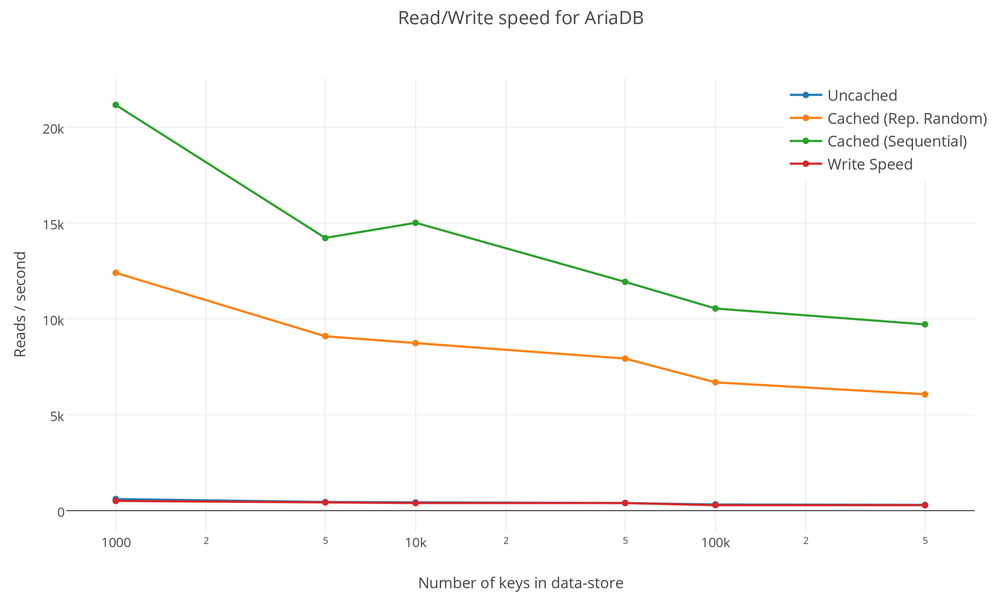

Figure 2: Read Write speed vs number of keys in the data store

Future Works

R/W Buffers: As of now, disk IOs happen each time the user tries inserting values, and since disk reads are time consuming; decreasing this disk IO with the help of flushing the buffer to disk would make the design more time efficient.

Compaction: After deletion of key-value pairs, there are much wasted space in the index tree, so re-indexing after a period of time will make this more space efficient.

Compression: By compressing the data written on files, disk space usage can be improved.

Distributed Datastore: The present implementation stores all the data in a single store, accessible to multiple clients. The datastore and the index can itself be distributed to balance the load.

References

[1] Bryan O' Sullivan et al. *Real World Haskell*. O'Reilly Media, Inc.

[2] Apache Software Foundation. Couchdb. `guide.couchdb.org/draft/btree.html`.

[3] Saurav Kumar. B+ tree implementation in c++. `github.com/2020saurav/contests`.

[4] Miran Lipovaa. Learn you a haskell. `learnyouahaskell.com`.